

Delving into the Linux boot process for an ARM SoC

Ajay Kumar, Thiagu Ramalingam
FDS S/W solutions - Samsung Semiconductor India Research

#osummit

CONTENTS

- **ARMv8 SoC basic architecture**

- SoC internal memory and bootup

- **Bootloader**

- Setup and Initialize the RAM
- Copy images to main memory
- Decompressing the kernel image

- **Kernel image header**

- Kernel image header
- Prepare for Jumping into Kernel

- **Deciding CPU boot configuration**

- **Jumping into Kernel: primary_entry**

- **Arch Independent Kernel Starting Point**

- **Process 0**

- **The First Processor Activation**

- **setup_arch**

- **Scheduler Initialization**

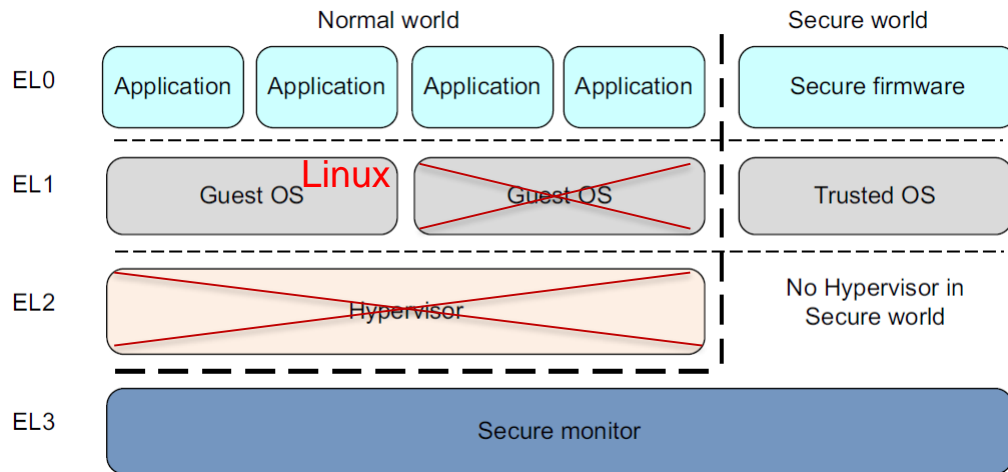
- **SMP on ARM SOC**

- **irq_init and time_init - System Timer**

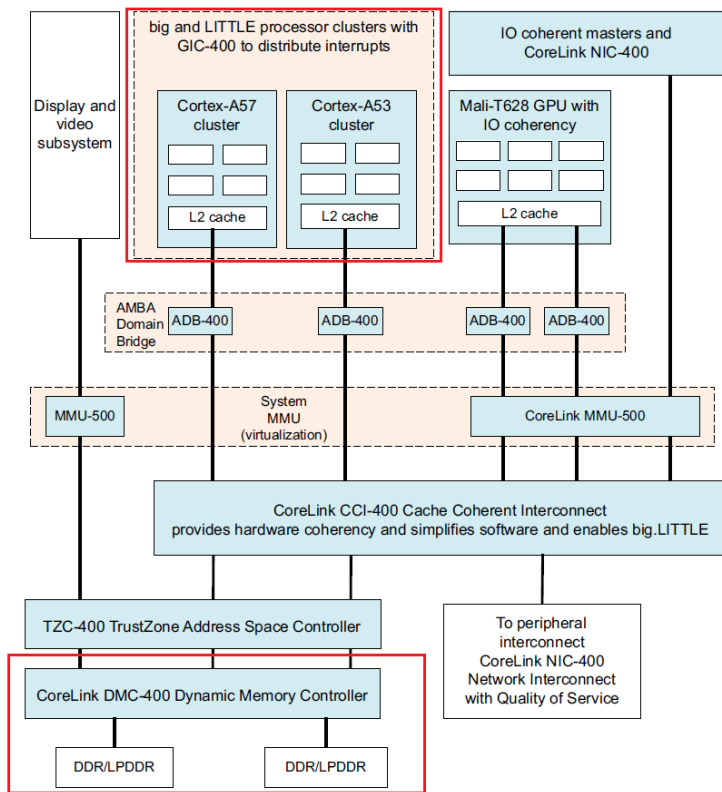
- **rest_init**

Assumptions

- **ARMv8 SoC**
- **Hypervisor not used**
- **BL0, BL1, etc - the bootloader “stages” conceptual only.**
- **Microcontrollers handling initial SoC boot aren’t covered**



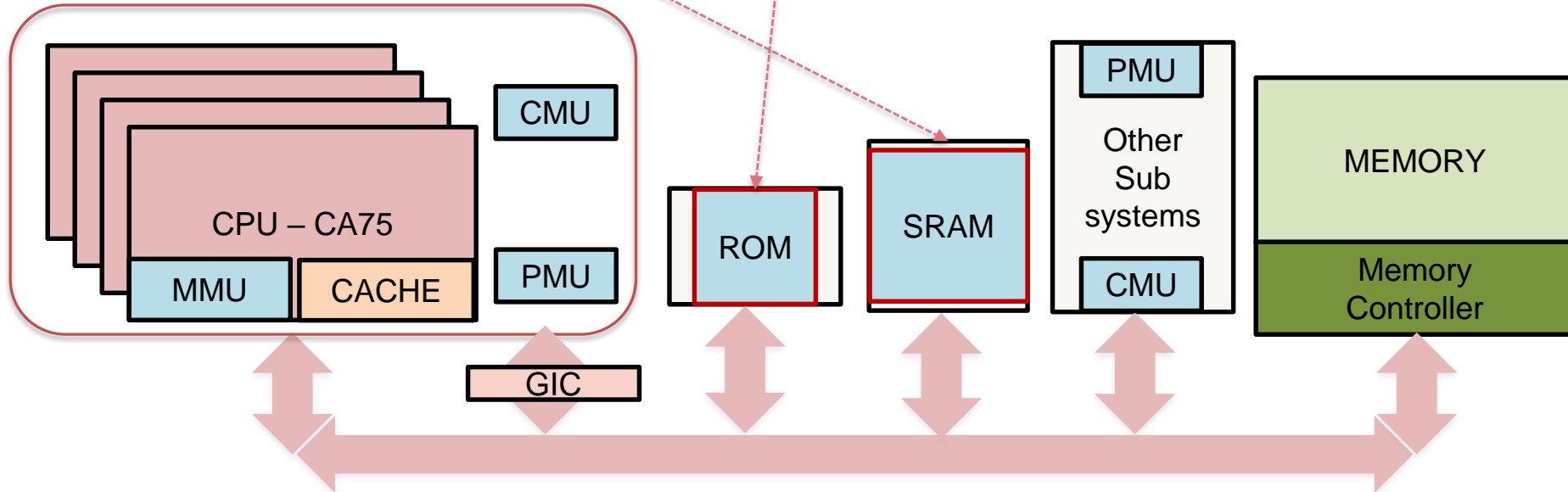
ARMv8 SoC basic architecture



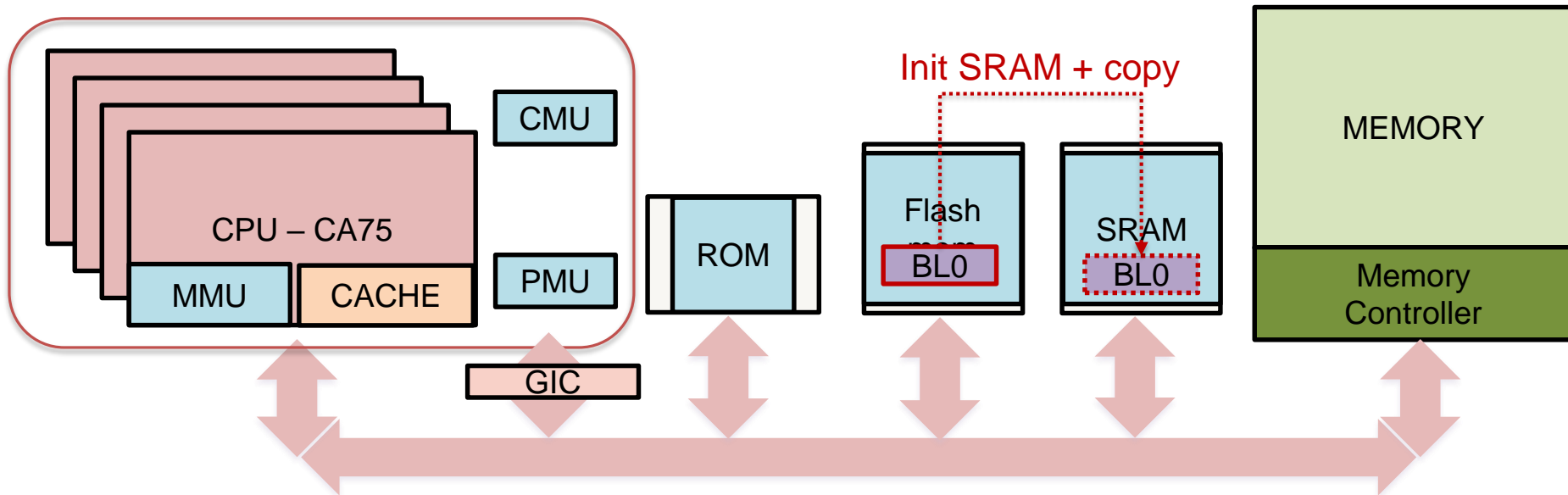
- Example of a simple (complex?) BigLittle ARM SoC:
- A SoC is basically an organization of various components:
 - CPU clusters
 - System buses
 - Memory controller
 - Main Memory
 - Other sub systems (Display, GPU, Peripherals, Host controllers, etc)
- The SOC also consists of components like Clock, Power switches, Power Domains for sub blocks.

SoC internal memory

- Apart from main memory, SoC will have a ROM (Read-Only Memory) which contains minimal code to setup the system for next stage binary loading. This piece of code is executed upon CA block reset.
- It might also have an SRAM (volatile memory) which can help in execution of initial C routines.

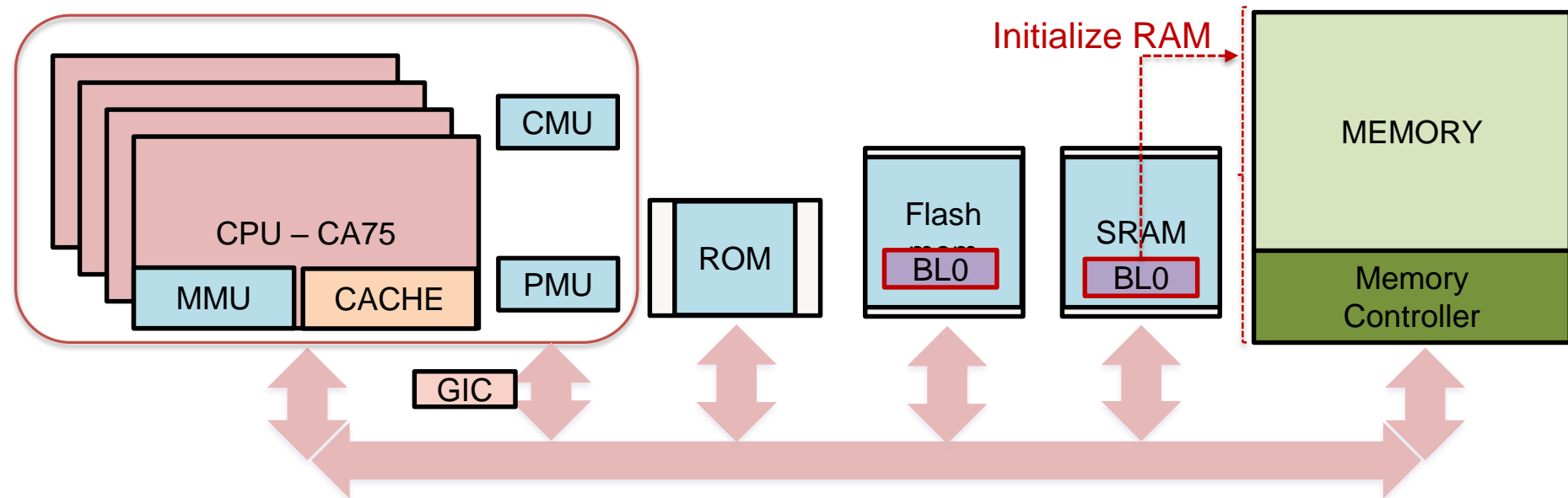


- The ROM code does minimal initialization of SRAM block and copy Bootloader(BL0) from storage/flash memory to SRAM memory.
 - Runs in EL3 mode
 - Interrupts are mostly disabled at this stage
 - Powering up core clocks, power domains
 - Setting up C environment on SRAM for BL0 execution



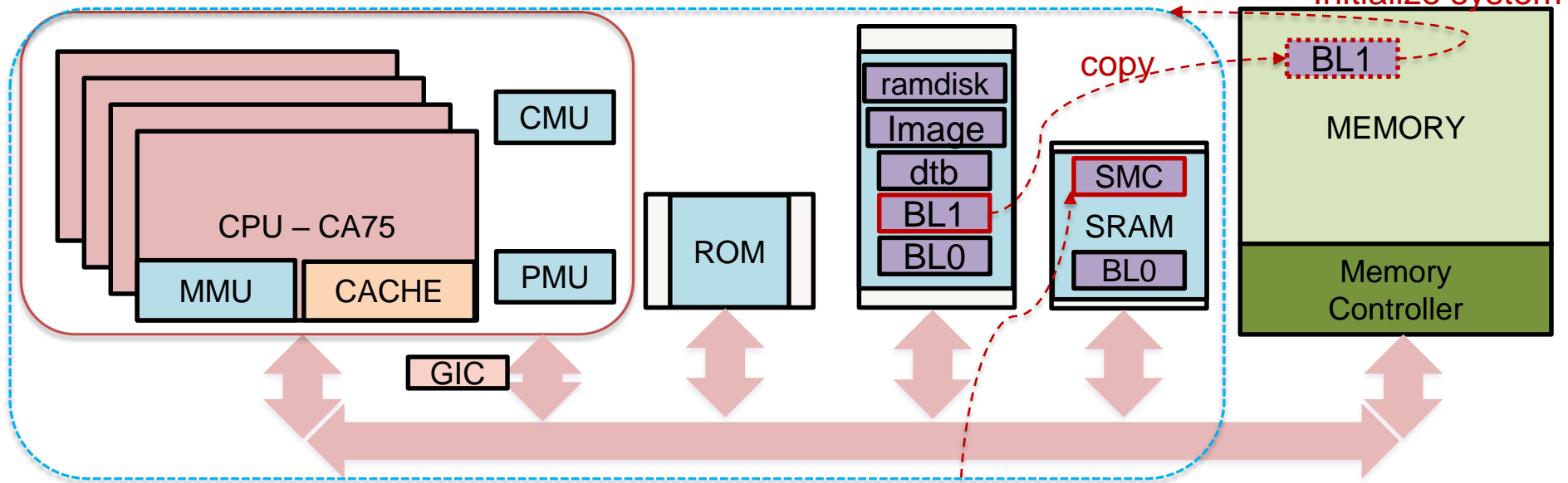
Setup and Initialize the RAM

- Now the Bootloader BL0 executing from SRAM can further initialize the system clocks, power domains and most importantly **initialize main memory**.
- The Bootloader is expected to find and initialize all RAM that the kernel will use. It performs this in a machine dependent manner.



BL1 and Secure Monitor

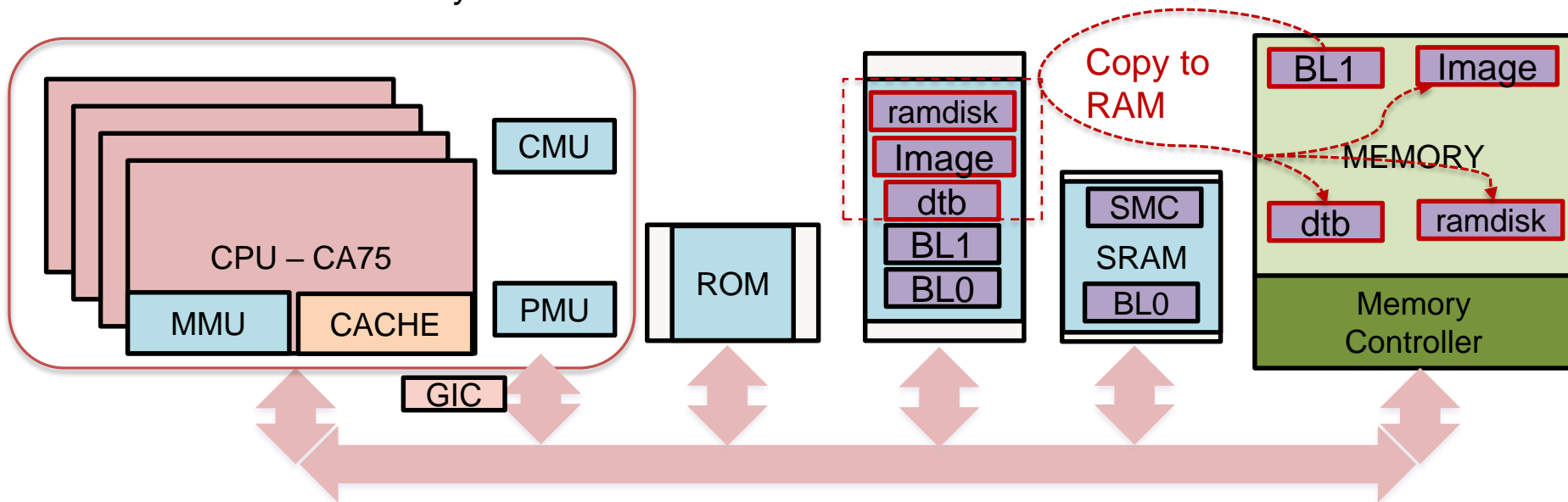
- Once the primary Bootloader BL0 has initialized the main memory, it can load a secondary bootloader (BL1) which can execute from main memory.
- BL1 initializes the system for supporting Linux boot, loads other binaries needed for Linux boot from storage to main memory. Can have interrupts enabled.



- BL0/BL1 should also keep a Secure Monitor code(SMC) for handling secure access.

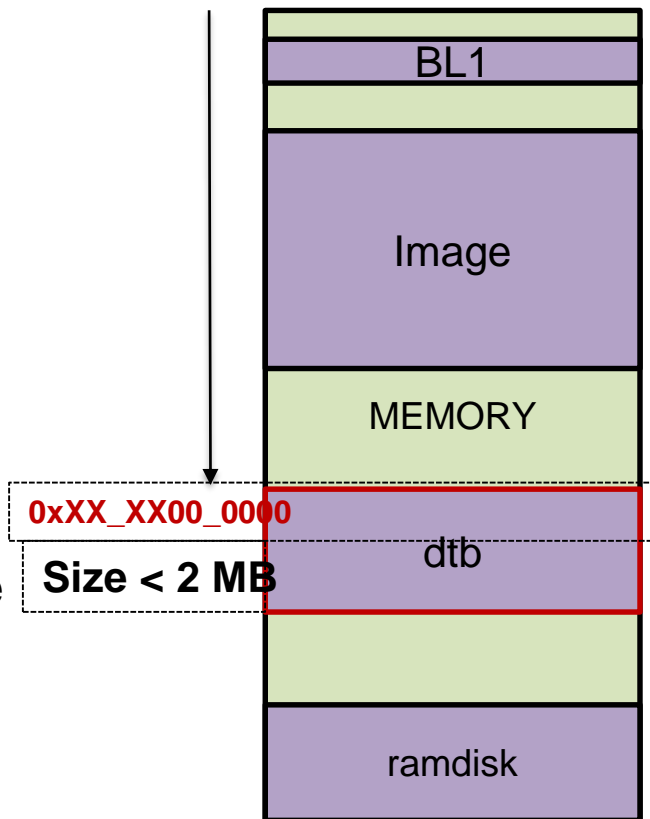
Copy images to main memory

- DTB – Device Tree Blob – Description of Hardware in Device Tree format
- Image – Actual Kernel binary
- Ramdisk – Initial RAMDISK – minimal rootfs loaded before mounting actual root file system. Required to execute init scripts
- All loaded to memory via BL1



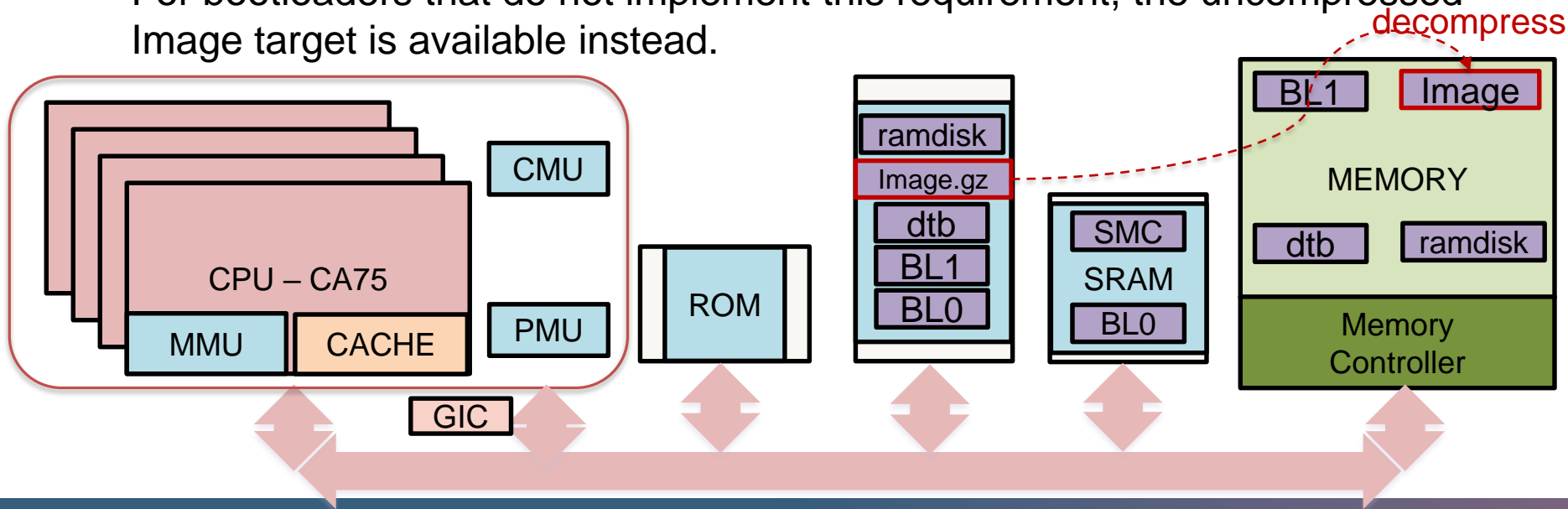
Device Tree Blob

- Description of Hardware in Device tree format
- Contains memory mapped addresses and information about CPU, memory, GPIO, clocks, peripherals, etc.
- Before the kernel is executed, bootloader selects proper device tree file and passes it as an argument to the kernel
- This is because the dtb will be mapped cacheable using blocks of up to 2 megabytes in size, it must not be placed within any 2M region which must be mapped with any specific attributes.



Decompressing the kernel image

- Image – Actual Kernel binary, Image.gz – Compressed Kernel binary
- The AArch64 kernel does not currently provide a decompressor and therefore requires decompression (gzip etc.) to be performed by the boot loader if a compressed Image target (e.g. Image.gz) is used.
- For bootloaders that do not implement this requirement, the uncompressed Image target is available instead.



The decompressed kernel image contains a 64-byte header as follows::

```
u32 code0;          /* Executable code */
u32 code1;          /* Executable code */
u64 text_offset;    /* Image load offset, little endian */
u64 image_size;     /* Effective Image size, little endian */
u64 flags;          /* kernel flags, little endian */
u64 res2 = 0;       /* reserved */
u64 res3 = 0;       /* reserved */
u64 res4 = 0;       /* reserved */
u32 magic = 0x644d5241; /* Magic number, little endian, "ARM\lx64" */
u32 res5;          /* reserved (used for PE COFF offset) */
```

- **code** – Start of text section
- **text_offset** – Obsolete
- **image_size** - Effective Image size
- Over the years, few of these fields have become obsolete (ex: text_offset)

Kernel image header flags

The decompressed kernel image contains a 64-byte header as follows::

```
u32 code0;          /* Executable code */
u32 code1;          /* Executable code */
u64 text_offset;    /* Image load offset, little endian */
u64 image_size;     /* Effective Image size, little endian */
u64 flags;          /* kernel flags, little endian */
u64 res2            = 0; /* reserved */
u64 res3            = 0; /* reserved */
u64 res4            = 0; /* reserved */
u32 magic           = 0x644d5241; /* Magic number, little endian, "ARM\<div>
```

Kernel Header dump

The decompressed kernel image contains a 64-byte header as follows::

```
u32 code0; /* Executable code */
u32 code1; /* Executable code */
u64 text_offset; /* Image load offset, little endian */
u64 image_size; /* Effective Image size, little endian */
u64 flags; /* kernel flags, little endian */
u64 res2 = 0; /* reserved */
u64 res3 = 0; /* reserved */
u64 res4 = 0; /* reserved */
u32 magic = 0x644d5241; /* Magic number, little endian, "ARM\<div>
u32 res5; /* reserved (used for PE COFF offset) */
```

- After placing the kernel image, what remains is setting up remaining environment for jumping into kernel.
- **Before jumping into the kernel:**
 - **Disable DMA capable devices** so that memory does not get corrupted
 - **CPU mode**
 - Primary CPU general-purpose register settings:
 - **x0 = physical address of device tree blob (dtb)** in system RAM.
 - x1 = 0, x2 = 0, x3 = 0
 - Secondary CPU general-purpose register settings:
 - x0 = 0, x1 = 0, x2 = 0, x3 = 0 (reserved for future use)
 - All forms of **interrupts must be masked** in PSTATE.DAIF (Debug, SError, IRQ and FIQ)
 - The **MMU must be off**.
 - **Caches:** The instruction cache may be on or off, and must not hold any stale entries corresponding to the loaded kernel image.

- **Before jumping into the kernel (contd...):**
 - **Architected timers:** Timers at different exception level have to be initialized.
 - **Coherency:** All CPUs to be booted by the kernel must be part of the same coherency domain on entry to the kernel. This may require IMPLEMENTATION DEFINED initialization to enable the receiving of maintenance operations on each CPU. For ARMv8 Linux, all CPU under SMP fall into same Inner Shareable domain.
 - **System registers:** All writable architected system registers at or below the exception level where the kernel image will be entered must be initialized by software at a higher exception level to prevent execution in an UNKNOWN state
 - The requirements described above for CPU mode, caches, MMUs, architected timers, coherency and system registers apply to **all CPUs**.
 - All CPUs must enter the kernel in the same exception level.

- The primary CPU jumps directly to the first instruction of the kernel image.
- The device tree blob passed by this CPU must contain an **'enable-method'** property for other cpu nodes.
- **"psci"** enable-method:
 - kernel will issue CPU_ON calls as described in Power State Coordination Interface
 - Secure monitor code (ATF) will take care of powering up CPU internally
 - Platforms mostly use PSCI method.
- **"spin-table"** enable-method:
 - must have a 'cpu-release-addr' property in their cpu node
 - These CPUs should spin outside of the kernel in a reserved area of memory polling their cpu-release-addr location
 - A wfe instruction may be inserted to reduce the overhead of the busy-loop and a sev will be issued by the primary CPU.

- Once the bootloader BL1 has performed all necessary SOC initialization (clocks, power domains) and prepared for jumping to kernel, it will jump to kernel.
- Lets take example of coreboot:

```
/* May update bl31_params if necessary. */  
void *bl31_plat_params = soc_get_bl31_plat_params(&bl31_params);  
  
/* MMU disable will flush cache, so passed params land in memory. */  
raw_write_daif(SPSR_EXCEPTION_MASK);  
mmu_disable();  
bl31_entry(&bl31_params, bl31_plat_params);  
die("BL31 returned!");
```

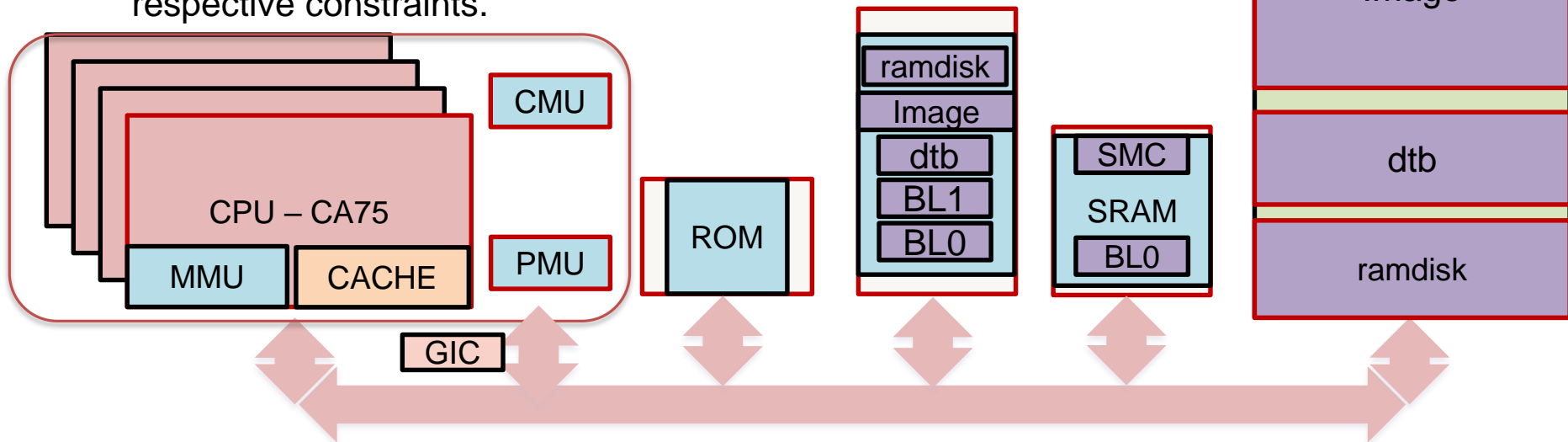
src/arch/arm64/arm_tf.c

```
/* Determine which image to execute next */  
image_type = bl31_get_next_image_type();  
  
/* Program EL3 registers to enable entry into the next EL */  
next_image_info = bl31_plat_get_next_image_ep_info(image_type);  
assert(next_image_info != NULL);  
assert(image_type == GET_SECURITY_STATE(next_image_info->h.attr));  
  
INFO("BL31: Preparing for EL3 exit to %s world\n",  
      (image_type == SECURE) ? "secure" : "normal");  
print_entry_point_info(next_image_info);  
cm_init_my_context(next_image_info);  
cm_prepare_el3_exit(image_type);
```

3rdparty/arm-trusted-firmware/bl31/bl31_main.c

Snapshot before jumping to kernel

- **CPU 0: EL1**
 - CMU, PMU – on
 - MMU – off
 - Data cache – off, Instruction cache – may be kept on
- Binaries placed in memory at respective addresses adhering to respective constraints.



head.S: primary_entry: Kernel entry point

```
/*
 * Kernel startup entry point.
 * -----
 *
 * The requirements are:
 *   MMU = off, D-cache = off, I-cache = on or off,
 *   x0 = physical address to the FDT blob.
 *
 * This code is mostly position independent so you call this at
 * __pa(PAGE_OFFSET).
 *
 * Note that the callee-saved registers are used for storing variables
 * that are useful before the MMU is enabled. The allocations are described
 * in the entry routines.
 */

__HEAD
/*
 * DO NOT MODIFY. Image header expected by Linux boot-loaders.
 */
efi_signature_nop // special NOP to identity as PE/COFF executable
b primary_entry // branch to kernel start, magic
.quad 0 // Image load offset from start of RAM, little-endian
le64sym _kernel_size_le // Effective size of kernel image, little-endian
le64sym _kernel_flags_le // Informative flags, little-endian
.quad 0 // reserved
.quad 0 // reserved
.quad 0 // reserved
.ascii ARM64_IMAGE_MAGIC // Magic number
.long .Lpe_header_offset // Offset to the PE header.

__EFI_PE_HEADER

__INIT

/*
 * The following callee saved general purpose registers are used on the
 * primary lowlevel boot path:
 *
 * Register Scope Purpose
 * x21 primary_entry() .. start_kernel() FDT pointer passed at boot in x0
 * x23 primary_entry() .. start_kernel() physical misalignment/KASLR offset
 * x28 __create_page_tables() callee preserved temp register
 * x19/x20 __primary_switch() callee preserved temp registers
 * x24 __primary_switch() .. relocate_kernel() current RELR displacement
 */
```



- **primary_entry** (or stext in earlier versions of linux kernel) is the entry point of arm64 architecture (arch/arm64/kernel/head.S)

arch/arm64/kernel/head.S: primary_entry

- preserve_boot_args**: Preserve the arguments passed by the bootloader in x0-x3 (x21 = x0 = FDT)
- init_kernel_el**: Setup based on the current kernel exception level - EL1/EL2 and return w0=cpu_boot_mode
- KASLR** (Kernel Address Space Layout Randomization) setting.

```
Sym_Code_Start(primary_entry)
bl    preserve_boot_args
bl    init_kernel_el           // w0=cpu_boot_mode
adrp  x23, __PHYS_OFFSET
and   x23, x23, MIN_KIMG_ALIGN - 1 // KASLR offset, defaults to 0
bl    set_cpu_boot_mode_flag
bl    __create_page_tables
/*
 * The following calls CPU setup code, see arch/arm64/mm/proc.S for
 * details.
 * On return, the CPU will be ready for the MMU to be turned on and
 * the TCR will have been set.
 */
bl    __cpu_setup              // initialise processor
b     __primary_switch
Sym_Code_End(primary_entry)
```

- set_cpu_boot_mode_flag**: Sets the __boot_cpu_mode flag depending on the CPU boot mode passed in w0, for later usage.
- __create_page_tables**: Setup the initial page tables
 - Identity mapping for MMU enable code (low address, TTBR0) – **idmap_pg_dir**.
 - Linear mapping for first few MB of the kernel – **init_pg_dir**

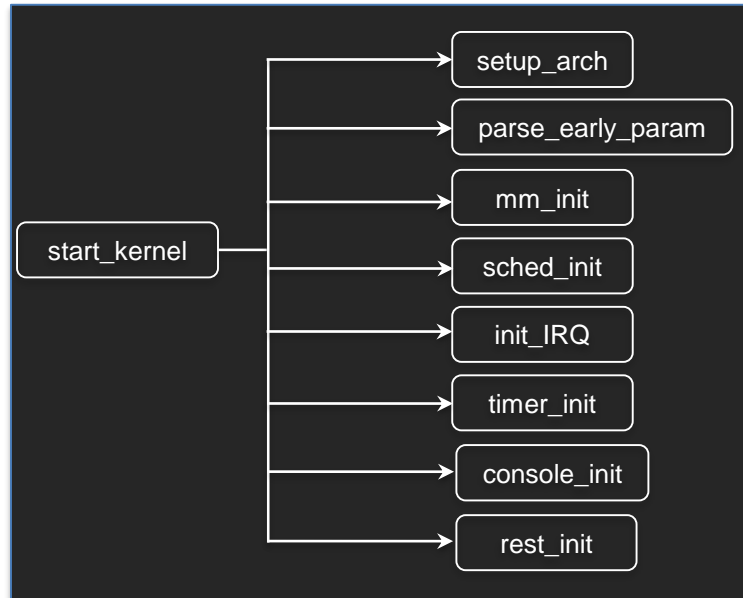


- **__cpu_setup:**
 - Initialize processor for turning the MMU on: clear TLB, set size for virtual, physical addresses, enable VM features.
 - Sets the TCR (Translation control register), and SCTRL (System control register) to do the same.

```
SYM_CODE_START(primary_entry)
    bl    preserve_boot_args
    bl    init_kernel_el          // w0=cpu_boot_mode
    adrp   x23, __PHYS_OFFSET
    and    x23, x23, MIN_KIMG_ALIGN - 1 // KASLR offset, defaults to 0
    bl    set_cpu_boot_mode_flag
    bl    __create_page_tables
/*
 * The following calls CPU setup code, see arch/arm64/mm/proc.S for
 * details.
 * On return, the CPU will be ready for the MMU to be turned on and
 * the TCR will have been set.
 */
    bl    __cpu_setup              // initialise processor
    b     __primary_switch
SYM_CODE_END(primary_entry)
```

- **__primary_switch:**
 - Set Page table address for TTBR0 (idmap_pg_dir), TTBR1(init_pg_dir)
 - **__enable_mmu** – check and configure for required Page granule, turn MMU on
 - Try to relocate kernel if possible - KASLR
 - call **__primary_switched:**
 - Assign EL1 vector table, Clear BSS, setup kernel stack, create FDT mapping, call **start_kernel**

- Kernel is always booted by architecture specific code. But then execution is passed to the `start_kernel` function that is responsible for common kernel initialization and is an architecture independent kernel starting point.
- The main purpose of the `start_kernel` is to finish kernel initialization process and launch the first init process.



```
asmlinkage __visible void __init start_kernel(void)
{
    char *command_line;
    char *after_dashes;

    set_task_stack_end_magic(&init_task);
    smp_setup_processor_id();
    debug_objects_early_init();

    cgroup_init_early();

    local_irq_disable();
    early_boot_irqs_disabled = true;

    /*
     * Interrupts are still disabled. Do necessary setups, then
     * enable them.
     */
    boot_cpu_init();
    page_address_init();
    pr_notice("%s", linux_banner);
    early_security_init();
    setup_arch(&command_line);
    setup_command_line(command_line);
    setup_nr_cpu_ids();
    setup_per_cpu_areas();
    smp_prepare_boot_cpu(); /* arch-specific boot-cpu hooks */
    boot_cpu_hotplug_init();

    build_all_zonelists(NULL);
    page_alloc_init();

    pr_notice("Kernel command line: %s\n", boot_command_line);
}
```

init/main.c

Kernel Creating Process 0

```
asmlinkage __visible void __init start_kernel(void)
{
    char *command_line;
    char *after_dashes;

    set_task_stack_end_magic(&init_task);
    smp_setup_processor_id();
    debug_objects_early_init();
}
```

```
struct task_struct init_task
#ifdef CONFIG_ARCH_TASK_STRUCT_ON_STACK
    __init_task_data
#endif
= {
#ifdef CONFIG_THREAD_INFO_IN_TASK
    .thread_info = INIT_THREAD_INFO(init_task),
    .stack_refcount = REFCOUNT_INIT(1),
#endif
    .state = 0,
    .stack = init_stack,
}
```

- `init_task` represents the initial task structure, that stores all the information about a process.
- The process 0 is statically defined. The only process that is not created by kernel thread nor fork.
- `set_task_stack_end_magic` function will set the stack border of `init_task`, which is the process 0.

The First Processor Activation

- The function initializes various CPU masks for the bootstrap processor.
- The processor id is got from the function:
 - `int cpu = smp_processor_id();`
- set the given CPU online, active, present, possible
 - `set_cpu_online(cpu, true);`
 - `set_cpu_active(cpu, true);`
 - `set_cpu_present(cpu, true);`
 - `set_cpu_possible(cpu, true);`
- `cpu_possible` : set of CPU ID's which can be plugged in at any time during the life of that system boot
- `cpu_present` : represents which CPUs are currently plugged in
- `cpu_online`: represents subset of the `cpu_present` and indicates CPUs which are available for scheduling

```
/* Activate the first processor.
 */
void __init boot_cpu_init(void)
{
    int cpu = smp_processor_id();

    /* Mark the boot cpu "present", "online" etc for SMP and UP case */
    set_cpu_online(cpu, true);
    set_cpu_active(cpu, true);
    set_cpu_present(cpu, true);
    set_cpu_possible(cpu, true);

#ifdef CONFIG_SMP
    __boot_cpu_id = cpu;
#endif
}
```

setup_arch()

early_ioremap_init:

- for early users of early_ioremap(paddr, size)

setup_machine_fdt

- Parse 'bootargs' from DT 'chosen' node
- Parse Physical Memory base and size, added into memblock subsystem
- Parse Machine model

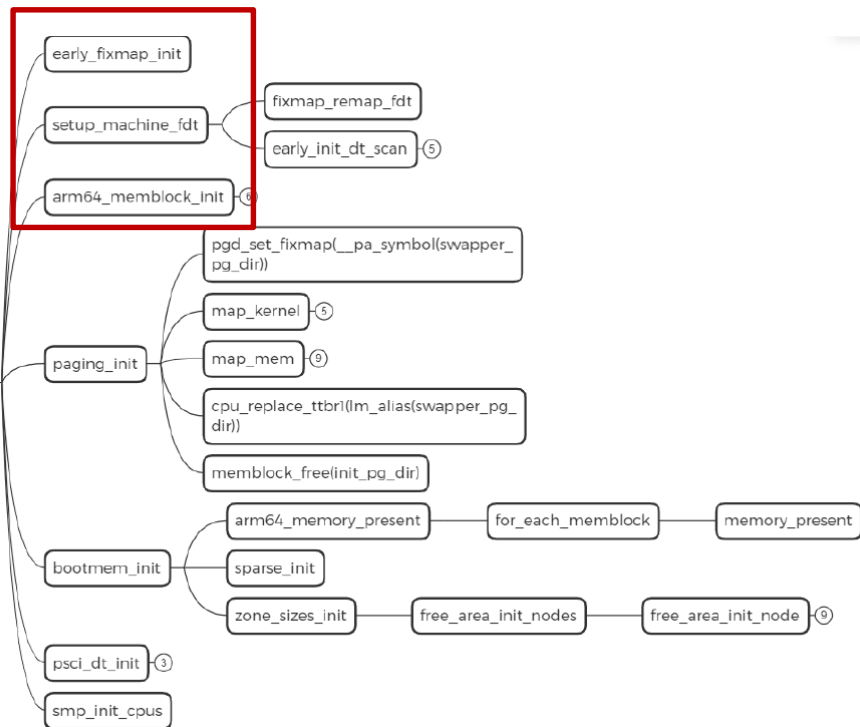
parse_early_param:

- early_param("mem", early_mem);
- early_param("earlycon", param_setup_earlycon);
- early_param("debug", debug_kernel);

cpu_uninstall_idmap: Remove idmap_pg_dir from TTBR0_EL1 and invalidate

arm64_memblock_init:

- Reserve memory used by kernel image
- Reserve memory specified in DT and specific initialization if any



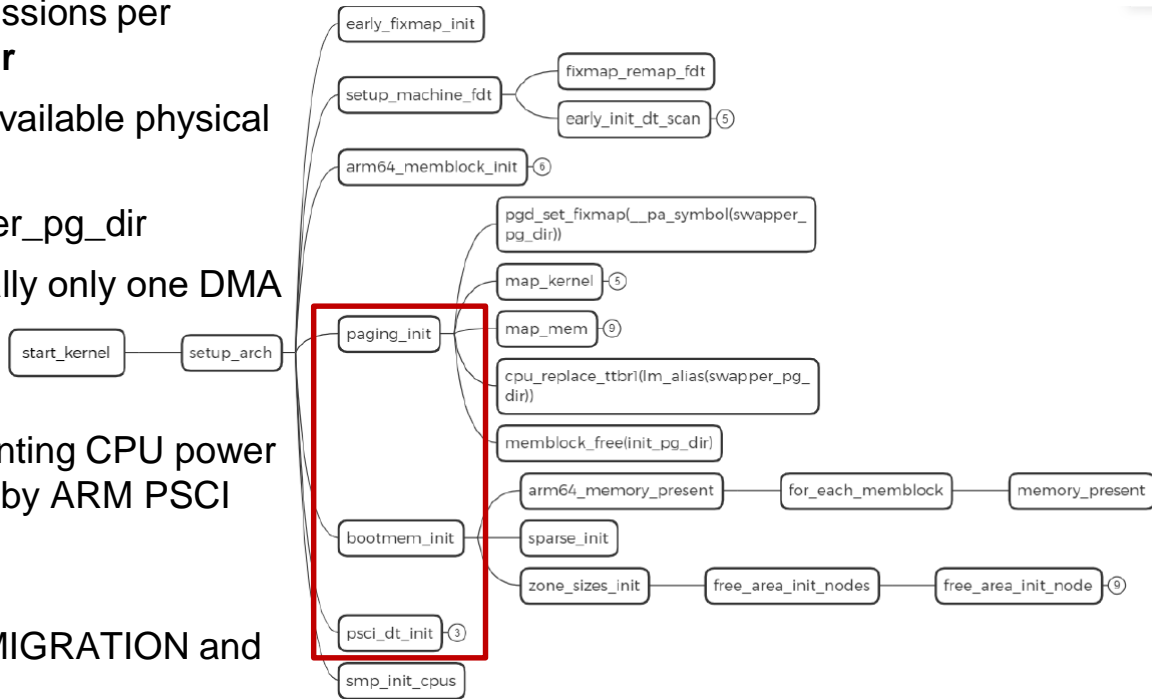
setup_arch() contd(...)

paging_init / bootmem_init

- Remap kernel sections `_text`, `_rodata`, `_data` and etc with fine grain permissions per segment to **swapper_pg_dir**
- Create Linear mapping for available physical memory blocks
- Switch page table to `swapper_pg_dir`
- Build memory zones – Usually only one DMA zone for ARM64

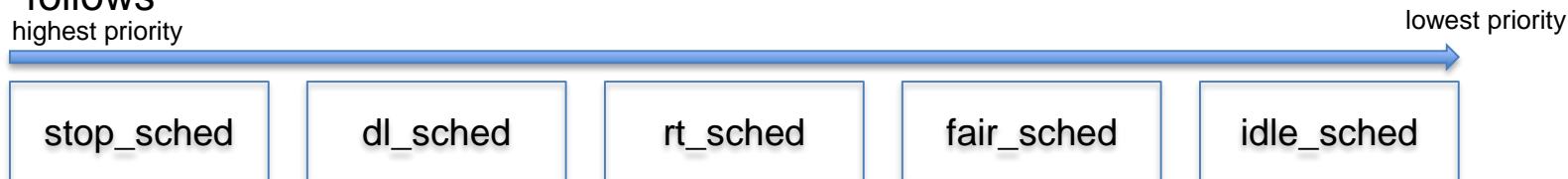
psci_init

- Firmware interface implementing CPU power related operations specified by ARM PSCI spec
- Including `CPU_ON/OFF/SUSPENDED/MIGRATION` and etc.



Scheduler Initialization

- The scheduler subsystem is one of the core subsystems of the kernel. It is responsible for the rational allocation of CPU resources in the system. It needs to be able to handle the scheduling requirements of complex different types of tasks.
- kernel has five scheduling classes, and the priority is distributed from high to low as follows



- Scheduling initialization located at start_kernel is relatively backward. At this time, the memory initialization has been completed, so you can see sched_init can already call kzmalloc and other memory application functions.
- sched_init initialize the run queue (RQ), the global default bandwidth of DL / RT, the run queue of each scheduling class, and CFS soft interrupt registration for each CPU.

A symmetric multiprocessor system (SMP) is a multiprocessor system with centralized shared memory called main memory (MM) operating under a single operating system with two or more homogeneous processors.

Most of the SMP code is not architecture dependent (in kernel directory).

Few SMP functions related to the SoC:

smp_init_cpus():

- Setup the set of possible CPUs (via `cpu_possible()`).
- Can be removed if the CPU topology is up to date in the device tree.
- Called very early during the boot process (from `setup_arch()`).

smp_prepare_cpus():

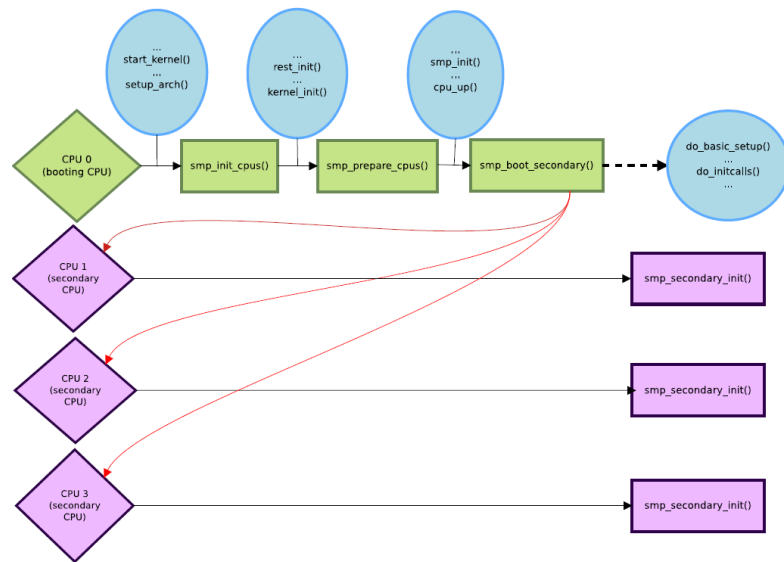
- Enables coherency.
- Initializes `cpu_possible` map.
- Prepares the resources (power, ram, clock...).
- Called early during the boot process (before the initcalls but after `setup_arch()`).

smp_secondary_init():

- Perform platform specific initialization of the specified CPU".
- Called from `secondary_start_kernel()` on the CPU which has just been started.

smp_boot_secondary():

- Actually boots a secondary CPU identified by the CPU number given in parameter.
- Called from `cpu_up()` on the booting CPU.



- **irq_init**
 - init_irq_stacks: Setup per CPU IRQ stack
 - irqchip_init → of_irq_init(__irqchip_of_table): Initialize the GIC controllers. Scans the device tree for matching interrupt controller nodes, and calls their initialization functions
- **time_init**
 - The function time_init() selects and initializes the system timer
 - It is a device that can be configured to periodically interrupt a processor with some predefined frequency.
 - One particular application of the timer, that it is used in the process scheduling
 - A scheduler needs to measure for how long each process has been executed and use this information to select the next process to run.
 - This measurement is based on timer interrupts.

- `start_kernel()` initializes dozens of kernel subsystems and ends calling **`rest_init()`**.
- `rest_init()` in its turn, spawns the very first user space process: **`kernel_init()`**.
- Its process id is 1 it will become the direct or indirect ancestor of all user space processes.
- It also spawns **`kthread process`** (normally, process id 2), that's the parent of all kernel threads.
- Finally, it runs **`cpu_idle()`**, a process that takes over the CPU whenever there is no other process using it.
- `kernel_init()` will start any additional CPU core.
- If there is a initial RAM disk is defined, it will decompress and mount it.
- Then, it load the device drivers, mount the root file system in read-only mode and finally call the init process (normally, in `/sbin/init`).

- For a complete understanding please refer to following spec:

- **References:**

- ARMv8 architecture: <https://developer.arm.com/documentation/den0024/a>
- Kernel source: <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/>
- Booting on ARM64: <https://www.kernel.org/doc/html/latest/arm64/booting.html>
- Analyzing Linux boot process : <https://opensource.com/article/18/1/analyzing-linux-boot-process>
- SMP boot in Linux : <https://developer.arm.com/documentation/den0013/d/Multi-core-processors/Booting-SMP-systems/SMP-boot-in-Linux>
- Memory Layout on AArch64 Linux : <https://www.kernel.org/doc/html/latest/arm64/memory.html>

THE LINUX FOUNDATION



**OPEN
SOURCE
SUMMIT**

NORTH AMERICA